Assignment 2: Processes and Threads
Student: Catalina Cojocaru

1 Assignment description

1.1 Process and Threads

In this assignment you have to create Linux processes and threads and synchronize their executions. Beside synchronizing their executions, your processes and threads will only display few messages and then terminate. The required synchronization will affect the order the messages will be displayed.

However, in order to be able to test automatically your solution, your processes and threads should call in appropriate places and order several functions we provide to you in the "*a2_helper.h*" and "*a2_helper.c*" files. In this sense, they should comply the following requirements:

1. The main process (the one created when you launch your program and illustrated in Figure 1 as P_1) must start by calling the "`init()`" function, whose role is to prepare your program to interact with our tester.

2. All processes, i.e. the main one and those created explicitly by your program, must call the "`info()`" function at their beginning and before their end respectively, with some predefined arguments, as illustrated in the box below, where you can see how the code of the main process and one of the processes it creates should look like.

Sample Process Code

```
int main(int argc, char **argv)
{
    // tester initialization
    // only one time in the main process
    init();

    // inform the tester about (main) process' start
    info(BEGIN, process_no, thread_no);

    // other process' actions
    ...

    // create a new process
    if (fork() == 0) {
        // inform the tester about process' start
        info(BEGIN, process_no, thread_no);
```

```
        // other process' actions
        ...

        // inform the tester about process' termination
        info(END, process_no, thread_no);
    }

    // inform the tester about (main) process' termination
    info(END, process_no, thread_no);
}
```

3. Each thread must call the "`info()`" function at its beginning and before its end respectively, with some predefined arguments, as illustrated in the box below.

Sample Thread Code

```
void thread_function(void* arg)
{
    // possible thread' actions
    ...

    // inform the tester about thread start
    info(BEGIN, process_no, thread_no);

    // other possible thread' actions
    ...

    // inform the tester about thread termination
    info(END, process_no, thread_no);
}
```

The "`info()`" function basically (at least from your point of view) displays a message. Its arguments have the following meaning:

1. the first one corresponds to one of the constant values defined in the "*a2_helper.h*" file, the only two supported being "`BEGIN`" and "`END`";
2. the second argument is an integer value, corresponding to a unique identifier of the calling process, as explained below;
3. the third argument is an integer value, corresponding to a unique identifier of the calling thread, as explained below.

1.2 Process and Thread Naming Convention

You program must create some number of processes as described below in Section 2. It must also associate that processes unique identifiers, independent of the unique process identifiers the operating system generates. Such, when required to generated N processes, the associated identifiers must be 1, 2, ...,

N. The way you associate these identifiers to created processes is described below. We will refer to processes in this document using labels derived from their associated identifiers, such P_1, P_2, ..., P_N for processes being associated identifiers 1, 2, ..., and N respectively.

Each process could, in its turn, be required to create some number of threads. When speaking about threads created by a process, you must pay attention that the implicit process' thread, i.e. the one implicitly started at process creation (so, not created explicitly with the new thread creation function) is not considered. Thus, if a process is said to create N threads, that means that it calls the new thread creation function N times.

Similar to processes, each process' threads are also associated unique thread identifiers (again, different by those associated by the operating system). Thus, for N created threads, the associated identifiers will be 1, 2, ..., N. The thread identifiers will be associated in a strict monotonically increasing manner, based on their creation order: first thread being associated the identifier 1, second thread being associated the identifier 2 and so on. The main thread is considered to be associated the thread identifier 0 (zero). Note that thread identifiers are unique only for threads of the same process, but not for two different processes.

In our requirements we will refer to a process' threads using labels similar to those for processes. However, while thread identifier ranges for different processes are not disjoint, a thread will be uniquely identified by the unique pair formed by its process' identifier and its own thread identifier. This can be noted in the way we specified arguments to the "`info()`" function. Correspondingly thread labels we use include both process and thread identifier, using the following naming convention: thread $T_{N.M}$ is the M's thread created by process N, where M' range is between 1 and the maximum number of threads created by that process.

Based on described identification and naming convention, the skeleton code of processes P_1 (the main one) and P_2 (one created process), considered to be run by their main threads (having identifier 0, i.e. threads $T_{1.0}$ and $T_{2.0}$ respectively) should look now like below.

Sample Code for Processes P_1 and P_2

```
int main(int argc, char **argv)
{
    // tester initialization
    // only one time in the main process
    init();

    // inform the tester about (main) process' start
    info(BEGIN, 1, 0);

    // other process' actions
    ...

    // create a new process
    if (fork() == 0) {
        // inform the tester about process' start
        info(BEGIN, 2, 0);
```

```
        // other process' actions
        ...

        // inform the tester about process' termination
        info(END, 2, 0);
    }

    // inform the tester about (main) process' termination
    info(END, 1, 0);
}
```

Similarly, the skeleton code of one of P_2's threads (e.g. $T_{2.5}$) should look like:

Sample Code for Thread $T_{2.5}$

```
void thread_function(void* arg)
{
    // possible thread' actions
    ...

    // inform the tester about thread start
    info(BEGIN, 2, 5);

    // other possible thread' actions
    ...

    // inform the tester about thread termination
    info(END, 2, 5);
}
```

1.3 Synchronization Mechanisms and Strategy

You are free to use any Linux synchronization mechanisms you want, like:

- System V semaphores,
- POSIX semaphores, or
- POSIX locks and condition variables.

You are even free to use a combination of the mentioned synchronization mechanisms, though we do not recommend something like this.

Regarding thread synchronization requirements described below, they are mainly expressed in terms of specific ordering criteria applied to the beginning and ending messages. You can control such ordering using the needed synchronization mechanisms before the corresponding calls to the "`info()`" function.

Other times the synchronization requirements are described in terms of a maximum number of threads that must be running in a certain process. Normally, this means that the main thread of that process should control the thread creation process, by using synchronization mechanisms. However, since our tester is only called by the "`info()`" function, you could also understand this requirement as the maximum number of threads running simultaneously

in their function between the BEGIN and END messages. In such a case, the created threads themselves could control how many of them are running (inside the thread function) in the same time, by using synchronization mechanisms. This kind of synchronization should be done before the threads call their "`info(BEGIN, ...)`" function.

The mentioned types of synchronization requirement should lead to the process and its thread skeleton functions looking like in the boxes below.

Sample Code for Processes P_1 and P_2 With Synchronization

```
int main(int argc, char **argv)
{
    // tester initialization
    // only one time in the main process
    init();

    // inform the tester about (main) process' start
    info(BEGIN, 1, 0);

    // other process' actions
    ...

    // create a new process
    if (fork() == 0) {
        // inform the tester about process' start
        info(BEGIN, 2, 0);

        // other process' actions
        ...

        // calls to synchronization mechanisms
        ...

        // new thread creation
        pthread_create(...);

        // inform the tester about process' termination
        info(END, 2, 0);
    }

    // calls to synchronization mechanisms
    ...

    // new thread creation
    pthread_create(...);

    // inform the tester about (main) process' termination
    info(END, 1, 0);
}
```

Sample Code for Thread $T_{2.5}$ With Synchronization

```
void thread_function(void* arg)
{
    // possible thread' actions
    ...

    // calls to synchronization mechanisms
    ...

    // inform the tester about thread start
    info(BEGIN, 2, 5);

    // other possible thread' actions
    ...

    // calls to synchronization mechanisms
    ...

    // inform the tester about thread termination
    info(END, 2, 5);
}
```

You can note that the same process could create both other new processes and threads, as it is the case of process P_1 in our sample code above. While at process creation everything in a parent process is copied in a new child process it creates, this also includes the possible threads that process started before. Though, in the child process only the parent's thread calling the "`fork()`" function will be active, the other ones not being scheduled at all. Still, because of the many complications that could occur, we recommend you to write the code of one process that is required to create both processes and threads such that it creates firstly the requested processes and only than the requested threads. This is actually what we tried to suggests you in the given sample code.

2 Assignment's requirements

Your are required to write a C program named "*a2.c*" by extending the provided stub, where the following requirements will be implemented.

2.1 Compilation and Running

Your C program must be **compiled with no error** in order to be accepted. A sample compilation command should look like the following:

Compilation Command

```
gcc -Wall a2.c a2_helper.c -o a2 -lpthread
```

Warnings in the compilation process will trigger a 10% penalty to your final score.

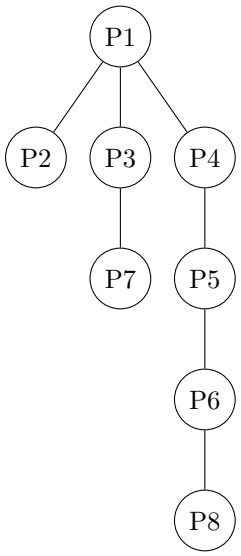

Figure 1: Required Process Hierarchy

When run, your resulted executable (we name it "*a2*") **must provide the minimum required functionality and expected results**, in order to be accepted. What such minimum means, will be defined below. The following box illustrates the way your program could be run.

Running Command

```
./a2
```

2.2 The process hierarchy

Your solution must generate the process hierarchy illustrated in Figure 1.

Each process having children must not terminate before its children. In terms of messages displayed by the "`info()`" function, the END message of a parent should never appear before any END message of its children.

2.3 Synchronizing threads from the same process

Process P2 must create 4 threads: T2.1, T2.2, ..., T2.4. You must impose on the execution of the process P2's threads the following synchronization conditions:

- Process' main thread, i.e. T2.0 must not terminate before the other 4 threads. This means that its "`info(END, ...)`" message must not be called before the similar calls of the other threads. In terms of messages, its END message must never be displayed before any END message of the other threads.
- Thread T2.2 must start before T2.3 starts and terminate after T2.3 terminates.

2.4 Threads barrier

Process P4 must create 40 threads: T4.1, T4.2, ..., T4.40. You must impose on the execution of the process P4's threads the following synchronization conditions:

- Process' main thread, i.e. T4.0 must not terminate before the other 40 threads.
- At any time, at most 6 threads of process P4 could be running simultaneously, not counting the main thread.
- Thread T4.14 can only end while 6 threads (including itself) are running.

2.5 Synchronizing threads from different processes

Process P6 must create 4 threads: T6.1, T6.2, ..., T6.4. You must impose on the execution of the processes P6's and P2's threads the following synchronization conditions:

- Process' main thread, i.e. T6.0 must not terminate before the other 4 threads.
- The thread T6.2 must terminate before T2.4 starts, but T6.3 must start only after T2.4 terminates.

3 User Manual

3.1 Self Assessment

In order to generate and run tests for your solution you could simply run the "*tester.py*" script provided with your assignment requirements. The script requires Python 3.x, not Python 2.x.

Sample Command for Tests Generation

```
python3 tester.py
```

Even if the script could be run in MS Windows OS (and other OSes), we highly recommend running it in Linux, while this is how we run it by ourselves, when evaluating your solutions.

The script will compile and run your program several times and assess your score based on the messages received from the program.

Restrictions:

- You are not allowed to interact with the file system in any way for this assignment.
- You are not allowed to use the following system calls and functions (nor to call them indirectly through other higher level functions):
  - `nanosleep()`,
  - `sleep()`,

  - `usleep()`,
  - `pthread_atfork()`.

- You are not allowed to communicate with the tester script in any other way excepting by calling the provided "`init()`" and "`info()`" functions.

When your assignment is graded, it is run inside a *docker* container. While a correct and deterministic solution will behave in the same way, a bug in your solution may pass unnoticed on your system but may cause a crash / undefined behavior during the "official" evaluation. For this reason, we encourage you to also test your solution using docker before submitting it. To do this, you need to:

- install *docker* on your machine (`sudo apt install docker.io`)
- create a Docker Hub account and login to it from the command line (it is needed for downloading the evaluation image)
- install the *docker* module for Python (`python3 -m pip install docker`)

To test your solution using the *docker* container, you need to provide the `--docker` argument to the testing script.

Sample Command for testing with Docker

```
python3 tester.py --docker
```

If you want to keep the docker container after running the tests (for instance, you may want to debug your code inside the container), use the option `--docker-persist`.

3.2 Evaluation and Minimum Requirements

- If your program has compilation warnings, a 10% penalty will be applied.
- If your program doesn't comply to the required coding style, a penalty up to 10% will be applied (decided by your instructor).

Do not cheat as we run plagiarism detection tools on all provided solutions.

Notes: The provided tests are not exactly the tests we will run on your provided solution. Besides, check on your code that the required functionality is completed and correctly provided as just running some set of tests does not necessarily cover all possible cases and prove the solution if perfect. It would thus be possible that some exceptional cases to be covered only be tests that we will run.